A Unified Execution Model for Data-Driven Applications on a Composable MPSoC

Ashkan Beyranvand Nejad
Delft University of Technology
Delft, The Netherlands
a.beyranvandnejad@tudelft.nl

Anca Molnos
Delft University of Technology
Delft, The Netherlands
a.m.molnos@tudelft.nl

Kees Goossens
Eindhoven University of Technology
Eindhoven, The Netherlands
k.g.w.goossens@tue.nl

***Abstract*—Multi-processor Systems on Chip (MPSoCs) execute multiple applications concurrently. These applications may belong to different domains, i.e., may have firm-, soft-, or non-real-time requirements. A composable system simplifies system design, integration, and verification by avoiding the inter-application interference. Existing work demonstrates composability for applications expressed using a single model of computation. For example, Kahn Process Network (KPN) and dataflow are two common data-driven parallel models of computation, each with different properties and suited for different application domains. This paper extends existing work with support for concurrent, composable execution of KPN and dataflow applications on the same MPSoC platform. We formalize a unified execution model by defining its operations that implement the different models of computation on the MPSoC, and discuss the trade-offs involved. Our experiments indicate that multiple applications modeled in KPN and dataflow run composably on an MPSoC platform.**

I. Introduction

In recent years the trend in consumer electronic devices is to execute an increasing number of applications simultaneously. Applications are functionally independent software units that are developed by possibly different parties. In consumer electronics, many applications are data-driven, i.e., streaming, such as audio and video codecs, or networking applications. They may have soft-, firm-, or non-real-time requirements. A Firm-Real-Time (FRT) application must never miss a deadline, whereas, a Soft-Real-Time (SRT) one may occasionally miss a deadline, and Not-Real-Time (NRT) ones are completely timing relaxed. Consequently, these three application domains require different design strategies and computation models.

Multi-Processor Systems on Chip (MPSoCs) are the platforms that can offer concurrent execution of multiple applications. *Composability* is proposed and advocated to alleviate system-wide, monolithic verification [1], [2] by avoiding inter-application interference. An MPSoCs is composable if an application's timing and functionality is not influenced by the behavior of other applications. Consequently, applications can be designed and verified in isolation and easily integrated on one platform, without invalidating their designed properties.

To fully exploit the computation power of an MPSoC, the parallelism is not restricted to the application level, but each application is further split in a number of concurrent tasks. Currently, most applications are still first designed and tested using an imperative, sequential, Model of Computation (MoC), typically C. If this first implementation follows a MoC subset denoted as *Nested Loop Programming* (NLP), an initial step towards a parallel implementation is made. Two of the most common parallel models of computation into which an NLP application can be automatically translated [3], are (i) *Kahn Process Network* (KPN) [4], and (ii) *dataflow* [5].

A *Model of Execution* (MoE) is a set of operations that are required to execute an application expressed in a MoC on an MPSoC. To the best of our knowledge, existing execution models are either tailored to a single model of computation [6]–[12] and do not provide insights into possible trade-offs in execution of each MoC.

In this context, the contributions of this paper are threefold. First, we propose one execution model that implements all of the NLP, KPN, and dataflow models of computation. Second, we discuss the trade-offs involved in mapping the computation models to the execution model. Third, we experimentally demonstrate that a set of applications modeled in NLP, KPN, and dataflow runs simultaneously, composably on an MPSoC platform designed following the template in [2], [13].

The rest of this paper is organized as follows. Related work is discussed in Section II. Section III gives an overview of the application computation models. Section IV introduces the target MPSoC platform. The model of execution is proposed and formalized in Section V for NLP, KPN, and dataflow. Section VI discusses the trade-offs involved in mapping the MoCs to the MoE. The experimental results is presented in Section VII, followed by the conclusions in Section VIII.

II. Related Work

To the best of our knowledge, none of the existing execution models supports simultaneous execution of multiple applications modeled as NLP, KPN, and dataflow. Moreover, unlike the existing work, our approach is composable. Various definition of composability exist [11], [14]. Our definition however is more restrictive in that inter-application interference is completely prohibited. The advantage is that a mix of FRT, SRT and NRT applications can be easily designed, verified, and integrated on the same MPSoC platform.

Several execution platforms for KPN applications were proposed [6]–[10], [15], [16]. The approach in [16] does not support multi-application execution. The platform in [15] executes KPN processes by scheduling them on reconfigurable accelerators. [14] proposes an MPSoC platform that supports multiple real-time applications. The system performance is estimated using the individual application timing profile and a model for the inter-application interference. Therefore, by our definition, all these approaches are not composable.

Application performance can be accurately analyzed using several dataflow models [17]–[19]. Thus dataflow is used to express real-time applications executed on MPSoCs [11], [12]. All these approaches allow the design of real-time applications, however the analysis requires bounds on the execution time of each task or preemption in bounded time. This is not generally the case for non-real-time applications, thus their integration on a common platform is not straightforward.

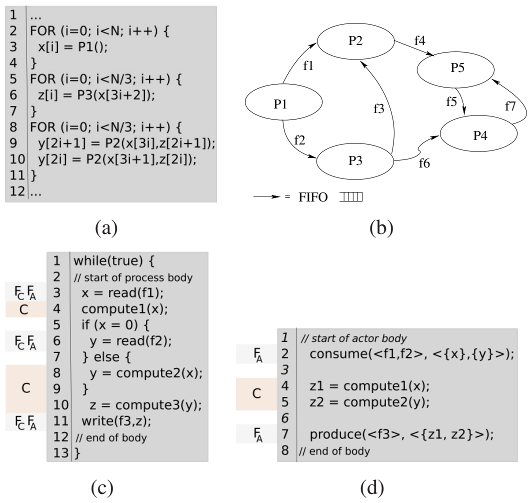
III. Application Models of Computation

Traditionally, applications are initially designed using a sequential MoC in a high level programming language, such as C. The Nested Loop Programming (NLP) is a subset of a sequential MoC. NLP models an application as a number of loops over single assignment basic *functions*, as presented in Figure 1(a). NLP can be automatically transformed into parallel MoCs [3], [20], e.g., KPN and dataflow. The KPN and dataflow models are networks of autonomous and concurrent *processes*, often referred to as *actors* in dataflow terminology [21]. A process corresponds to one or more function calls in the NLP model of the application, and it is a functional mapping from input streams to output streams (corresponding to the function's arguments). Each process executes for a possibly infinite number of activations. Processes synchronize by communicating packets of data, i.e., *tokens*, in a First-In-First-Out (FIFO) order, along unidirectional channels, as presented in Figure 1(b). In what follows, for clarity, we use the term *process* for KPN and *actor* for dataflow.

A KPN process body, illustrated in Figure 1(c), consists of a sequence of `read`, `compute`, and `write` operations. These operations may be interleaved in any order, and a process may read or write an arbitrary number of tokens from or into a FIFO. Each FIFO implementation has bounded capacity [8]. A process blocks on a read or write when the FIFOs does not have enough input data or output space, respectively.

A dataflow actor body is a sequence of `consume`, `compute`, and `produce` operations, in this strict order, as presented in Figure 1(d). A firing rule specifies, for one actor activation, for each incoming and outgoing edge, the number of input tokens consumed and the number of tokens produced, respectively. Once the firing rule is satisfied, an actor executes its entire body without blocking. Different variants of dataflow models exist, e.g., Static Dataflow (SDF), Cyclo-Static Dataflow (CSDF) [17], Variable Rate Dataflow (VRD) [18], some of which are analyzable. This means that existing formalisms [19] can derive an end-to-end latency and throughput of an application, given the worst case timing of each of its actors. In this paper, we consider only the analyzable dataflow model variants and we specifically focus on CSDF.

KPN and dataflow have different properties that make them suitable for different application domains. Dataflow is suitable for the FRT domain that demands timing analysis. However,



```
...
FOR (i=0; i<N; i++) {
  x[i] = P1();
}
FOR (i=0; i<N/3; i++) {
  z[i] = P3(x[3i+2]);
}
FOR (i=0; i<N/3; i++) {
  y[2i+1] = P2(x[3i],z[2i+1]);
  y[2i] = P2(x[3i+1],z[2i]);
}
...
```

```
while(true) {
// start of process body
  x = read(f1);
  compute1(x);
  if (x = 0) {
    y = read(f2);
  } else {
    y = compute2(x);
  }
    z = compute3(y);
  write(f3,z);
// end of body
}
```

```
// start of actor body
  consume(<f1,f2>, <{x},{y}>);

  z1 = compute1(x);
  z2 = compute2(y);

  produce(<f3>, <{z1, z2}>);
// end of body
```

Fig. 1. Application computation models: (a) a Nested Loop Programmed (NLP) application, (b) an application task graph, (c) a KPN process example code, (d) a dataflow actor example code

dataflow cannot model highly dynamic application behaviour, e.g., the production and consumption of an arbitrary number of tokens on a channel. Such behavior is common in the signal processing domain, e.g., variable-length encoding and decoding. KPN is a suitable model for such dynamic applications, as it allows arbitrary production and consumption rates and arbitrary interleaving of communication and computation inside a process. However, KPN is not amenable to timing analysis, thus it can only fit NRT and SRT applications. Essentially, execution of both KPN and dataflow MoC on a MPSoC platform broadens the supported application domains.

IV. Target Platform

Applications execute on an MPSoC platform, which consists of a hardware and a software infrastructure. Here, we target a *composable* platform [2], [13]. In this section, we briefly introduce the hardware and software infrastructure.

MPSoC Hardware Architecture: The hardware infrastructure comprises processor and memory tiles interconnected via a Network-on-Chip (NoC) [22]. A processor tile consists of a processor, local memory, and Remote Direct Memory Access modules (RDMA) [23]. All these resources are virtualized to achieve application isolation and therefore system composability.

MPSoC Software Architecture: The software executing on our MPSoC has two layers, namely the application and the Real-Time Operating System (RTOS) layer. An *application* consists of a set of *tasks* executing an infinite number of iterations and communicating via FIFOs. Each task iteration is a sequential set of execution operations, as detailed in the next section. We define a task's status as *eligible* if it can execute, meaning that it has enough data and space in the input and output FIFOs, respectively. If a task is not eligible, its status is *blocked*.

The RTOS provides an interface to the MPSoC resources, meaning that (i) it offers an Application Programming In-

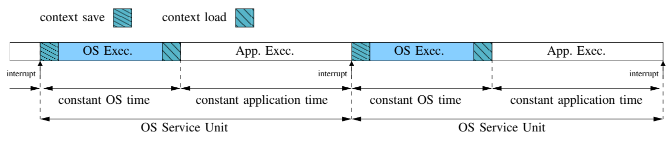


Fig. 2. A composable RTOS operational time-line

terface (API) for accessing the MPSoC's resources, e.g., communication channels, memories, peripherals, and (ii) it schedules applications and tasks on the processor. Here only the inter-task communication APIs are discussed, as the other APIs, e.g., memory management, are similar with what a conventional RTOS would offer. For more detailed discussion on the RTOS realization, we refer to [13].

The RTOS assigns processor time in basic quanta of fixed duration denoted as "application slots", implemented by a timer issuing interrupts. Between two consecutive application slots there is an "OS slot" in which the RTOS performs context switching, monitoring, handles timer interrupts, and schedules the next application to run. The processor time-line is presented in Figure 2. The set of all application slots is denoted as application time, and similarly, the set of all OS slots is denoted as OS time. The processor scheduling has two hierarchical levels: (i) inter-application, and (ii) intra-application, i.e., task scheduling. The processing time is shared between applications following a strict *time division multiplexing* (TDM) policy, in which each application is allocated a fixed set of slots. Thus the OS prohibits applications to take hold of each other's processor time. Moreover, the communication APIs are safely implemented such that applications do not overwrite each other's data, and do not monopolize resources, e.g., communication channels, RDMAs, etc.. Therefore, the targeted RTOS is composable.

V. Model of Execution

A *Model of Execution* (MoE) defines the set of operations involved in the execution of an application on an MPSoC. An application is expressed in a model of computation, e.g., KPN, dataflow, An MoC defines operations for *computation* and *communication*, e.g., `read`, `produce`, thus these should have equivalents in the execution model. Moreover, the RTOS provides the application *scheduling* operations. Therefore, the execution operations can be categorized as: (i) computation, (ii) communication, and (iii) scheduling operations. The implementation of the MoC using the execution operations are represented with a regular language sequence.[1] We first abstract from scheduling and present the set of execution operations and how they implement to each of the NLP, KPN and CSDF models. Then, we detail the scheduling for each of these MoCs.

A computation operation, C, is defined as the sequence of instructions that implements the task's functionality between two consecutive communication operations. Inter-tasks communication requires FIFO administration and access execution operations. A FIFO administration operation is a space- or data-*check* denoted with F_C. A FIFO buffer *access*, i.e., placing/retrieving data into/from the buffer, is represented as F_A. F_A may be performed only after making sure that space or data exist, via F_C.

The NLP model is mapped on the target execution platform as a simple application with one task and no FIFOs. KPN processes and CSDF actors are each implemented as tasks, and the inter-process and inter-actor communication is implemented through FIFOs. A KPN process activation or CSDF actor firing corresponds to a task iteration. The similarity between these two models stops here. A CSDF actor starts only if its firing rule is satisfied, its corresponding task's status therefore is valid for an entire iteration. A KPN process may immediately start (is eligible), and it blocks whenever it executes a `read` or `write` for which there is not enough data and/or space. Once it has started an iteration, a KPN task is not guaranteed to finish it without blocking, thus the task status is not valid for an entire iteration.

KPN `read` and `write` operations require both FIFO check and FIFO access, whereas the CSDF `produce` and `consume` operations require only FIFO access. Formally, `read` and `write` operations are implemented as $F_C^{[1,\infty)}F_A$ (where F_C^{∞} models that the task may wait for data infinitely), and `consume` and `produce` operations as F_A. In the KPN model the `read` and `write` operations may be arbitrarily interleaved with `compute`, $(C + F_C^{[1,\infty)}F_A)^{[0,\infty)}$. Here, F_C^{x} represents checking a FIFO x times where $x \in [1, \infty)$. $(C+F_C^{[1,\infty)}F_A)^{\infty}$ models the infinite firings of a process. In CSDF the execution order is strict, starts with all `consume` operation, i.e., the first $F_A^{[0,N]}$, continues with all `compute`s, C, and ends with all `produce` operation, the last $F_A^{[0,N]}$. These correspondence of each MoC's operation with an execution operation is also illustrated in Figure 1(c) & 1(d) beside the KPN and CSDF pseudo code lines.

Each level of the processor scheduling, i.e., inter-application and intra-application, is implemented with a separate scheduler. The inter-application scheduler selects only the application that owns the next slot. This execution operation is denoted with S_A. Each application may utilize its own task scheduling policy to determine which of its tasks will run in its slots. The task scheduler may be executed in the OS or in the application time. We define S_T as the execution operation that selects a task according to a given policy, e.g., TDM and Round-Robin. In KPN any policy can be used to schedule a task although scheduling an eligible tasks is more reasonable. Thus scheduling a KPN task can be implemented as an S_T operation. In CSDF the task scheduler has to find an eligible task, thus it selects a task, S_T, and a check of each of its FIFOs, $F_C^{[0,N]}$, repeatedly, formally resulting in $(S_T F_C^{[0,N]})^{[1,N]}$. If an eligible task is not found, the idle task is scheduled.

Table I presents the operations executed in OS time and application time for NLP, KPN, and CSDF. For KPN and CSDF we detail the cases in which the task scheduler is performed in: (i) OS time and (ii) application time. The

[1]If A and B are two operations, the regular expression language is defined as follows: $A\epsilon = \{A\}$, $(A + B) = \{A, B, AB\}$, $A^2 = \{AA\}$, $A^{[1,\infty)} = \{A, AA, AAA, ...\}$, and $A^{[0,N]} = \{\epsilon, A, AA, ..., A^N\}$.

underlined operations in the table indicate the body of tasks, i.e., processes or actors. Composability requires the RTOS to always assign a fixed number of slots to an application, thus the S_A operation is always executed in the OS slot.

TABLE I
COMPOSABLE EXECUTION MODEL FOR MPSOC
TASK SCHEDULING IN OS-TIME (I) AND APPLICATION-TIME (II)

MoC	OS Time	Application Time
NLP	S_A	C
KPN (i)	$S_A S_T$	$(\underline{C + F_C^{[1,\infty)} F_A})^{[1,\infty)}$
(ii)	S_A	$(S_T(\underline{C + F_C^{[1,\infty]} F_A})^{[0,\infty)})^{[0,\infty)}$
CSDF (i)	$S_A(S_T F_C^{[0,N]})^{[1,N]}$	$(\underline{F_A^{[0,N]} C F_A^{[0,N]}})^{[0,1]}$
(ii)	S_A	$((S_T F_C^{[0,N]})^{[1,N]}(\underline{F_A^{[0,N]} C F_A^{[0,N]}})^{[0,1]})^{[0,\infty)}$

The execution model of KPN with tasks scheduled in OS time, KPN (i), follows directly from the models of the task and the task scheduler. In this case in each OS slot the application and task are selected, $S_A S_T$, and in each application slot the task is executed, $(C + F_C^{[1,\infty)} F_A)^{[0,\infty)}$.

The execution model for KPN with tasks scheduled in the application time, KPN (ii), is more complex. In the application slot, task selection has to be repeated whenever a task iteration has finished, or the task is blocked. In detail, after a task is initially selected, i.e., S_T in Table I, the task may `compute`, C, or `read` or `write`. In case of `read` or `write`, the FIFO has to be first checked, F_C. If the check fails, the current task is blocked, thus instead of polling for FIFO data or space, another task is selected, i.e. starts from the beginning by executing another S_T. Otherwise, the check returns successfully and the FIFO buffer is accessed, F_A. After this access, another FIFO may be read or written, thus the procedure may be repeated. Furthermore, after a task iteration finishes, another task is selected according the same algorithm above.

In CSDF, the execution model is a composition of $F_A^{[0,N]} C F_A^{[0,N]}$, as the task body, and $(S_T F_C^{[0,N]})^{[0,N]}$, as the task scheduling, The latter may be placed in OS time, CSDF (i), or in application time, CSDF (ii). If the task scheduler cannot find an eligible task, in CSDF (i), it schedules the idle task, $(F_A^{[0,N]} C F_A^{[0,N]})^0$, and in CSDF (ii), it continues polling for an eligible task, $(S_T F_C^{[0,N]})^{[0,N]}(F_A^{[0,N]} C F_A^{[0,N]})^0$.

VI. TRADE-OFFS IN EXECUTION MODELS

A designer has many choices to implement an application on a platform. Two important ones are: (i) which model of computation to use and (ii) where to execute the task scheduling. Table II summarizes the trade-offs between the possible combinations of these two choices.

Unlike KPN, CSDF is analyzable, the status of a task is *constant during an iteration*, and it cannot model dynamic applications. The analyzability is explicit in the CSDF's MoEs in Table I. The operations in the CSDF task body and scheduler execute for a bounded number of repetitions ($[0, N]$), in contrast to KPN where an infinite number of executions is possible ($[0, \infty]$). Moreover, no F_C operation exist in a CSDF task's body. Thus the task status is constant during an iteration and it can be determined by checking the firing rules. Hence a scheduler can make use of this information to increase processor utilisation. KPN can models dynamic behavior, i.e., the order if `read`s and `write`s and the number of tokens accessed is arbitrary. This behavior is not analyzable, thus KPN does not suit FRT applications. Moreover the status of a task is not available before giving the control to that task, leaving room for less scheduling optimizations.

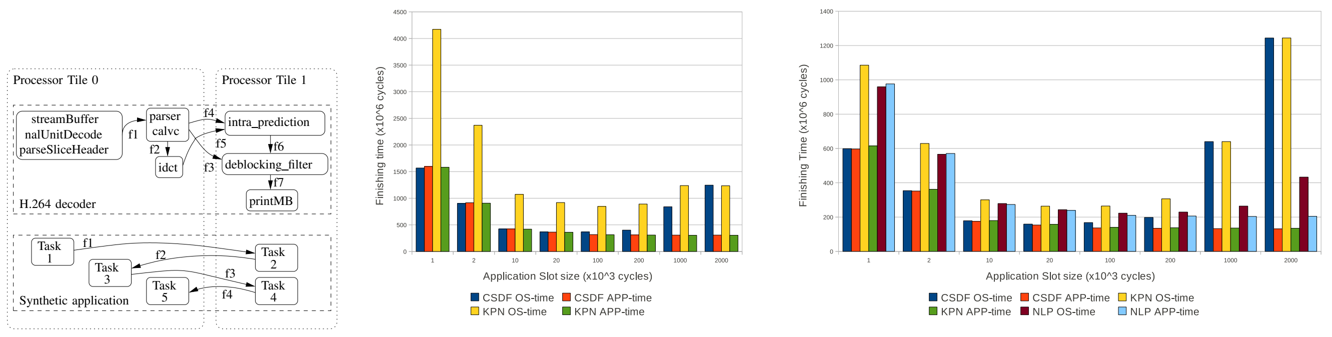
In case the task scheduler is executed in the OS time, all scheduling decisions are taken exclusively in the OS slot. When a task finishes or it is blocked before its slot depletes, the remaining time is wasted. Thus this approach is *non-work-conserving*, potentially leading to a lower processor utilisation. When executed in the application time, a new task may be scheduled immediately after a blocked or finished task. Therefore the entire application time can be utilized, i.e. this method is *work-conserving*. Moreover, a task scheduling policy is supported only under the condition that it is thoroughly verified and characterized, as the worst case RTOS execution time should be tightly bounded. While this is a requirement for a real-time application, it is not necessary for non-real-time applications, where it can limit the available options. The limitation of task scheduling in application time is that applications do not have access to timers and interrupts, unless these are virtualised, hence on our platform the scheduler policy has to be *cooperative*, i.e., cannot preempt tasks.

TABLE II
TRADE-OFFS SUMMARY

Appl. Model	OS time scheduling	Application time scheduling
NLP	FRT, SRT, NRT	FRT, SRT, NRT
KPN	SRT, NRT	SRT, NRT
	variable status during an iteration	variable status during an iteration
	preemptive	cooperative
	non-work conserving	work conserving
	strictly verified scheduler	any scheduler
CSDF	FRT, SRT, NRT	FRT, SRT, NRT
	preemptive	cooperative
	constant status during an iteration	constant status during an iteration
	non-work conserving	work conserving
	strictly verified scheduler	any scheduler

VII. CASE STUDY

In this section we study the composability of the proposed execution model, followed by a performance investigation of different MoCs mapped to the MoE. The target platform has two processing tiles and a memory, communicating via an on-chip interconnect. The MPSoC is implemented on a Virtex 6 FPGA; all the MPSoC resources run at clock frequency of 50 MHz. This platform executes two applications, a synthetic one and H.264 video decoder. The synthetic application is a parallel application with five tasks, as presented in Figure 3(a). The H.264 decoder is initially modelled in NLP, which is then parallelized in six tasks, in two versions, one for KPN and one for CSDF. We execute this applications concurrently using Round-Robin task scheduler in the OS time and application time.



(a) (b) (c)

Fig. 3. (a) Synthetic and H.264 application on a 2-Tile MPSoC Platform; finishing time for 20 iterations of (b) synthetic application, and (c) H.264 decoder

To study the composability, we have executed applications in two scenarios, in isolation and concurrently. The applications timing results, i.e., execution and finishing times, showed no differences in the two scenarios, indicating that the system is composable.

We consider the finish time of the last task, i.e. *Task 5* in synthetic application and *printMB* in H.264, as the metric for the applications performance. Given that we always start the applications execution at the same point in time, the smaller the finish time of the last task, the better is the performance. The performance of the synthetic application is illustrated in Figure 3(b) for various application slot sizes. Except for the small slot sizes, the task scheduling in application time leads to better performance, in both KPN and CSDF, because it is work-conservative and utilizes the entire application slot. The bigger the application slot size is, the more application time is wasted by the OS time scheduling policy. For small slot sizes the overhead caused by the frequent context switch between slots leads to relativelly poor performance, and here for CSDF the performance differences between OS and application time scheduling are minor. In the case of KPN, OS time scheduling performs poorly regardless of the slot size, because the status of a task is not known when the task is selected and when the task is blocked the entire application slot is wasted.

Figure 3(c) illustrates the performance of the H.264 modeled in NLP, KPN and CSDF. NLP performs better than CSDF with OS time scheduling for large application slot size, because the wasted time in CSDF exceeds the benefits of parallelizing the application. Similar to the synthetic application, in H264 application time scheduling leads to better performance, small slot sizes have large overhead, and the KPN with OS time scheduling has the worst performance.

VIII. Conclusions

In this paper we propose a unified execution model to implement data-driven applications on a composable MPSoC platform. The applications can be realized in three different models of computation, e.g., Nested Loop Programmed (NLP), Kahn Process Network (KPN), and dataflow. The execution model is formalized by introducing the operations necessary to execute each of these models on an MPSoC. We discuss the trade-offs involved in executing these models such as: (i) which model of computation to use for an application, (ii) who should schedule the tasks of the application (the operating system or the application itself) and (iii) the properties of different scheduling options, e.g., preemptive and work-conserving. Using the proposed execution model, we experimentally study the system composability and performance of a synthetic and a H.264 video decoder applications, modeled as NLP, KPN and dataflow, executed on an MPSoC prototyped in FPGA.

References

[1] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, 1997.
[2] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken, "CoMPSoC: A template for composable and predictable multi-processor system on chips," *ACM ToDAES*, 2009.
[3] A. Turjan *et al.*, "Translating affine nested-loop programs to process networks," in *CASES*, 2004.
[4] G. Kahn, "The semantics of a simple language for parallel programming," in *IFIP*, 1974.
[5] E. A. Lee *et al.*, "Dataflow process networks," 2002.
[6] J. Y. Hur *et al.*, "Systematic customization of on-chip crossbar interconnects," in *ARC*, 2007.
[7] J. Hur *et al.*, "Customizing reconfigurable on-chip crossbar scheduler," in *ASAP*, 2007.
[8] T. Stefanov *et al.*, "System design using Kahn Process Networks: The Compaan/Laura approach," in *DATE*, 2004.
[9] C. Zissulescu *et al.*, "Laura: Leiden architecture research and exploration tool," in *FPL*, 2003.
[10] W. Haid *et al.*, "Efficient execution of Kahn Process Networks on multi-processor systems using protothreads and windowed FIFOs," in *ESTIMedia*, 2009.
[11] A. Kumar *et al.*, "Analyzing composability of applications on MPSoC platforms," *J. Syst. Archit.*, vol. 54, March 2008.
[12] O. Moreira *et al.*, "Multiprocessor resource allocation for hard-real-time streaming with a dynamic job-mix," in *RTAS*, 2005.
[13] H. Hansson *et al.*, "Design and implementation of an operating system for composable processor sharing," *Microprocessors and Microsystems*, 2011, special issue on Network-on-Chip Architectures and Design Methodologies.
[14] J. Castrillon *et al.*, "Trace-based KPN composability analysis for mapping simultaneous applications to MPSoC platforms," in *DATE*, 2010.
[15] M. Dyer *et al.*, "Efficient execution of process networks on a reconfigurable hardware virtual machine," in *Field-Programmable Custom Computing Machines (FCCM)*, 2004.
[16] I. Auge *et al.*, "Platform-based design from parallel C specifications," *TCAD*, 2005.
[17] T. M. Parks *et al.*, "A comparison of synchronous and cycle-static dataflow," in *Asilomar*, 1995.
[18] M. Wiggers *et al.*, "Efficient computation of buffer capacities for multi-rate real-time systems with back-pressure," in *CODES*, 2006.
[19] M. Bekooij *et al.*, "Efficient buffer capacity and scheduler setting computation for soft real-time stream processing applications," in *SCOPES*, 2007.
[20] B. Kienhuis, E. Rijpkema, and E. Deprettere, "Compaan: deriving process networks from matlab for embedded signal processing architectures," in *CODES*, 2000.
[21] J. Eker *et al.*, "A structured description of dataflow actors and its application," 2003.
[22] K. Goossens *et al.*, "The Aethereal network on chip after ten years: Goals, evolution, lessons, and future," in *DAC*, 2010.
[23] B. Akesson *et al.*, "Architectures and modeling of predictable memory controllers for improved system integration," in *DATE*, 2011.